Reordering Tests for Faster Test Suite Execution

Reordering Tests for Faster Test Suite Execution

Panagiotis Stratis, Ajitha Rajan
School of Informatics, University of Edinburgh

ACM Reference Format:
Panagiotis Stratis, Ajitha Rajan. 2018. Reordering Tests for Faster Test Suite Execution. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195048

1 INTRODUCTION

As software takes on more responsibility, it gets increasingly complex, requiring an extremely large number of tests for effective validation [1, 6]. Executing these large test suites is expensive, both in terms of time and energy. Cache misses are a significant contributing factor to execution time of software. In this paper, we propose an approach that helps order test executions in a test suite in such a way that instruction cache misses are reduced, and thereby execution time.

We conduct an empirical evaluation with 20 subject programs and test suites from the SIR repository, EEMBC suite and LLVM Symbolizer, comparing execution times and cache misses with test orderings maximising instruction locality versus a traditional ordering maximising coverage, as well as random permutations.

Performance gains were considerable for programs and test suites where the average number of different instructions executed between tests was high. We achieved an average execution speedup of 6.83% and a maximum execution speedup of 17% over subject programs with differing control flow between test executions.

2 APPROACH

To maximise temporal re-use of instructions across test executions, we minimise the distance between tests, where the distance between two tests, T_i and T_j, is defined as:

$$D(T_i, T_j) = \frac{\#basic\text{-}blocks\ different\ between\ T_i\ and\ T_j}{Total\ \#basic\text{-}blocks\ visited\ by\ all\ tests} \quad (1)$$

We define this distance metric based on the idea that temporal instruction locality would be enhanced if subsequent tests execute as many common instructions as possible. For scalability reasons, we use basic blocks instead of instructions. Using this distance metric, we perform nearest neighbour analysis in order to produce a test permutation where the distance between subsequent tests is minimized. Our approach is illustrated in figure 1: The first step is to dynamically analyse test executions and map them to their set of visited basic blocks. We then compute the distance matrix and perform nearest neighbour analysis. The starting test is the one with the most unvisited neighbours.

This optimisation algorithm, presented in [7], is quadratic in complexity with respect to number of tests. The main computational bottleneck is the calculation of the distance matrix which has $\frac{N^2}{2}$ complexity for a test suite with N tests. We found in our evaluation that the optimisation algorithm was unable to scale beyond 14K tests.

To allow the optimisation to be scalable, we implemented an approximate nearest neighbour algorithm which builds a multi-probe locality-sensitive hashing (LSH) index [3] instead of calculating the full distance matrix. LSH is a technique for grouping points in high-dimensional space into buckets based on some distance measure (in our case the hamming distance). Our approximation algorithm operates on the same input as the original algorithm but instead of computing the distance matrix, we construct a multi-probe LSH index. A starting test case is picked at random and the order is created using approximate nearest neighbour queried from LSH index until the index is empty.

2.1 Test Analysis.

For mapping each test to the set of its visited basic blocks we used Intel's Pin tool [2]. Pin is an instrumentation-based dynamic analysis framework which allows the development of customized dynamic program analysis tools (a.k.a Pintools). We developed a Pintool that records visited basic blocks for a program execution. Given a C/C++ program and its corresponding tests, our implementation will execute each test independently and dynamically analyse it with our Pintool.

2.2 Approximate Nearest Neighbour.

For locality sensitive hashing we used the C++ implementation of FLANN [4], a library for performing fast approximate nearest neighbour in high dimensional spaces. In our configurations, we had 12 hash tables, each with a key length of 20.

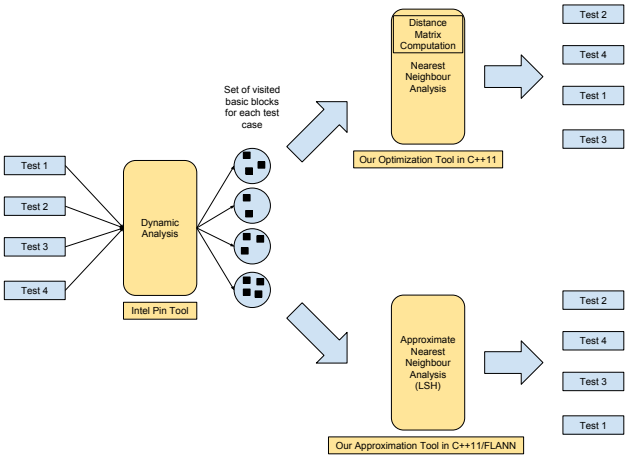

Figure 1: Implementation

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195048

3 EXPERIMENT RESULTS

We conduct our experiments over programs from different application domains to assess the execution performance of our test case orderings. For each benchmark, we use four different types of test suites and we measure their execution time:

- **Opt**- Test suite ordered according to our optimisation algorithm.
- **Approx** - Test suite ordered using our approximation algorithm.
- **BC** - Test suite ordered greedily by the branch coverage metric.
- **Random** - We randomly permute the tests in the test suite. We generate 2000 random permutations per benchmark.

Furthermore, we profile each permutation with *Cachegrind* that is part of Valgrind [5] in order to measure instruction cache miss rate.

Figure 2 shows the results for one fo the benchmarks, LLVM Symbolizer. The histogram frequencies for the 2000 random permutations, and 100 runs of each of *Opt, Approx* and *BC* are shown. LLVM Symbolizer showed significant execution speedup with both *Opt* (17%) and *Approx* (13%) relative to *BC*. Furthermore, `Opt` outperformed 98% of the *Random* permutations, while *Approx* outperformed 88%. Median *Opt* performance was better than median *Approx* by 4%.

The premise in locality orderings (*Opt* and *Approx*) is that they will reduce the number of instruction cache misses by increasing cache locality. This in turn will translate to faster, or reduced, execution time. We checked this premise for both *Opt* and *Approx* orderings by measuring instruction cache miss rates. We find that the reduction in execution times closely follows reduction in cache misses, for the optimised orderings, relative to *BC*, over the subject programs.

We also measured overhead incurred in ordering tests for *Opt* and *Approx*. Overhead of `Opt` is quadratic in size of test suite and does not scale beyond 14K tests for EEMBC programs. Overhead of `Approx` is considerably smaller and scales well to large test suites (70K tests for EEMBC). We found overhead could be further reduced with the use of GPUs. It is worth noting that, ordering algorithms can be performed offline and overhead need not be incurred during actual test suite execution.

Lastly, we find that average test distance is positively correlated with *Approx* execution time improvement ($r = 0.76$), as illustrated in figure 3. A higher average test distance indicates the differences in instructions executed by tests in the test suite is higher. Ordering for instruction cache locality has a higher impact on performance gains for such test suites since it ensures that tests wich execute too many different instructions between them are not executed in succession, avoiding cache misses that result from the difference. LLVM Symbolizer has the highest average TD among subject programs of 42% and also the highest speedup with `Approx` of 13%.

4 CONCLUSION

We presented an approach for ordering tests to increase cache locality across test executions. We conducted empirical evaluations to assess execution speedups using the original approach and approximation relative to random orderings and a greedy ordering for branch coverage. We used programs from SIR, EEMBC benchmarks and an LLVM Symbolizer.

Our evaluations revealed that ordering test executions to maximise instruction locality reduces cache misses and speeds up execution.

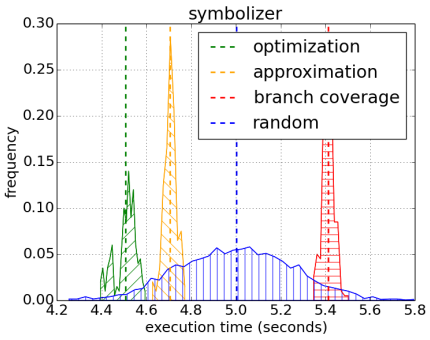

Figure 2: Histogram frequencies of execution time for `Opt`, `Approx`, `BC`, `Random` Test Suites for LLVM Symbolizer

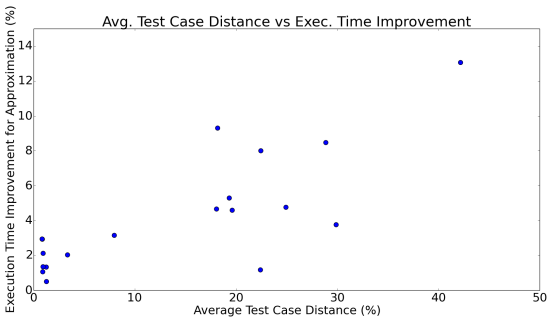

Figure 3: Test Distance versus Time Improvement for Approx ordering over BC

Based on the results over our subject programs, we recommend the `Approx` ordering of tests in a test suite since it achieves (1) Comparable execution speedups to `Opt`, and (2) Scales well to large numbers of tests, as opposed to `Opt`. Overhead of `Approx` is less than that of `Opt` for large test suites and can be further reduced by running the algorithm on GPUs. For subject programs whose executions fit in the cache, we found average test distance serves as a good guide for determining whether `Approx` ordering will result in reasonable performance improvements.

REFERENCES

[1] Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats PE Heimdahl. 2016. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Transactions on Software Engineering and Methodology* 25, 3 (2016), 25.

[2] Chi-Keung Luk et al. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[3] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*. 950–961.

[4] Marius Muja and David G Lowe. 2012. Fast matching of binary features. In *Computer and Robot Vision (CRV), 2012 Ninth Conference on*. IEEE, 404–410.

[5] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. 89–100.

[6] Ajitha Rajan. 2006. Coverage metrics to measure adequacy of black-box test suites. In *Automated Software Engineering, 2006. 21st IEEE/ACM International Conference on*. IEEE, 335–338.

[7] Panagiotis Stratis and Ajitha Rajan. 2016. Test case permutation to improve execution time. In *Automated Software Engineering, 2016 31st IEEE/ACM International Conference on*. IEEE, 45–50.